Property Access Traces for Source Incremental Model-to-Text Transformation

Babajide Ogunyomi, Louis M. Rose, and Dimitrios S. Kolovos

Department of Computer Science, University of York
Deramore Lane, Heslington, York, YO10 5GH, UK.
[bjo500, louis.rose, dimitris.kolovos]@york.ac.uk

Abstract. Automatic generation of textual artefacts (including code, documentation, configuration files, build scripts, etc.) from models in a software development process through the application of model-to-text (M2T) transformation is a common MDE activity. Despite the importance of M2T transformation, contemporary M2T languages lack support for developing transformations that scale with the size of the input model. As MDE is applied to systems of increasing size and complexity, a lack of scalability in M2T (and other) transformation languages hinders industrial adoption. In this paper, we propose a form of runtime analysis that can be used to identify the impact of source model changes on generated textual artefacts. The structures produced by this runtime analysis, property access traces, can be used to perform efficient source-incremental transformation: our experiments show an average reduction of 60% in transformation execution time compared to non-incremental (batch) transformation.

1 Introduction

Although MDE can reduce systems complexity and increase developer productivity [1], achieving scalability of MDE processes, practices and technologies remains an open research challenge and is important for widespread industrial adoption [2]. The scalability challenges in MDE are numerous, and include: performant persistence of very large models, modularity and reusability in the definition of very large modelling languages, and efficient propagation of change between artefacts (including models). This paper focuses on the latter challenge, in the context of propagating changes from models to textual artefacts (such as source code, documentation, or build scripts).

Our primary motivation for this work stemmed from our participation in an EC FP7 project (INESS, grant #218575) which involved applying model-to-text transformation to generate code that was amenable to model checking. Code generation from UML models supplied by our industrial partners took about 1 hour. Re-generation of code took 1 hour to execute even for small changes to the source model, because all code files were being re-generated even when the changes did not affect the content of some of them. Ideally, the execution

time of the code-generating transformation would have been directly proportional to the magnitude of the change to the source model: small changes to the model would have resulted in significantly reduced execution time of the code-generating transformation. This ideal is realised with a *source incremental* transformation engine [3].

In this paper, we propose *property access traces*, an approach to achieving source incremental model-to-text (M2T) transformation. Property access traces use runtime analysis to capture information about the way in which a transformation accesses its source models. When the source models change, a property access trace provides an efficient means for determining which subset of the transformation must be re-executed to propagate changes to the textual artefacts. Crucially, a property access trace allows the transformation engine to reduce (and ideally eliminate) execution of the parts of the transformation that have not been affected by the changes to the source models, and the M2T transformation scales better as a whole. This paper makes the following contributions:

- A design for computing and querying property access traces in order to perform efficient propagation of changes from models to textual artefacts (Section 3).
- An implementation of property access traces for a contemporary M2T transformation language, EGL [4], including a discussion of its limitations (Sections 3 and 4).
- An empirical evaluation and discussion of the benefits of property access traces for two existing M2T transformations (Section 4).

2 Background: M2T Transformation

This section briefly summarises contemporary approaches to M2T transformations and the different types of incrementality that are needed for effective and efficient M2T transformation.

The majority of contemporary M2T transformation languages use an approach (Listing 1.1), in which M2T transformations comprise several modular templates, whose structure closely resembles the generated text. Any portions of generated text that vary over model elements are replaced with *dynamic (executable) sections*, which are evaluated with respect to one or more source models. Any portions of generated text that remain the same are termed *static sections*. A M2T transformation normally comprises several templates, and co-ordination logic that invokes each template on the relevant part of the source models.

```
1 Hello, [%= person.name %]!
```

Listing 1.1. A template-based M2T transformation, in EGL syntax, which contains a static section ("Hello, "), a dynamic section (that outputs the value of the name attribute of a *person* model element) and another static section ("!").

Incrementality in model transformation – and in general – seeks to react to changes in an artefact (such as a model) in a manner that minimises the need for redundant computations. For M2T transformation, three types of incrementality have been identified: user edit-preserving incrementality, target incrementality, and source incrementality [3]. User-edit preserving incrementality and target incrementality are now widely supported, but source incrementality is not [5]. In this paper, we focus on source incrementality and argue that it is an essential feature for providing scalable M2T transformation capabilities.

Source incrementality is the capability of a M2T transformation engine to respond to changes in its source models in a way that minimises (and ideally eliminates) the need for re-computations that will not eventually have an impact on its output. Our intuition, which we investigate and assess in this paper, is that achieving a high degree of source incrementality can significantly improve the efficiency of complex transformations, especially when they operate on large or complex source models (e.g., with many cross-references between model elements and/or inter-dependencies between source models).

3 Property Access Traces

In this section, we propose *property access traces*, which contain concise and precise information collected during the execution of a M2T transformation and can be used to detect which templates need to be re-executed in response to a set of changes in the input model(s). We demonstrate how property access traces can provide comprehensive support for source incrementality for contemporary template-based M2T transformation engines. In contrast to existing approaches to source incremental model-to-text and model-to-model transformation, property access traces do not rely on model differencing or static analysis (which can be computationally expensive and imprecise).

This section provides an overview of using property access traces for source incremental transformation, discusses the way in which existing template-based M2T languages can be extended with support for property access traces, and briefly describes a prototypical implementation of property access traces for the EGL [4] M2T language.

3.1 Overview

To provide support for source incrementality, a transformation engine must be capable of identifying the subset of the transformation that is sensitive to changes in its input models (impact analysis), and re-executing the subset of the transformation to update the target (change propagation). Performing accurate impact analysis presents arguably the greatest challenge: in a template-based M2T transformation, a template might be sensitive to some types of change to a model element, but not to others. In the example presented in Figure 1, student reports are generated by a template that, clearly, is sensitive to changes to the name of a course (e.g., "SEPR" changes to "Software Project"), but not to the name of

the lecturer (e.g., "Mary Johnson" changes to "Mary Johnson-Smith"), similarly, changes to student names should not trigger re-generation of course reports.

Property access traces, as discussed below, provide a lightweight but effective mechanism for recording an M2T transformation's execution information which can be then used to detect relevant changes in the source model, and to determine which parts of the transformation need to be re-executed against which model elements. When a transformation is first executed, property access traces are captured and persisted in non-volatile storage. A *property access trace* records which parts of the transformation access which parts of the source models. In subsequent executions of the transformation, the property access trace is used to detect whether the source models have changed, and to re-execute only those parts of the transformation that are affected by source model changes. Determining which parts of the transformation to re-execute is possible because we require that transformation templates have two characteristics: they must be stateless and deterministic. A stateless template takes its data only from input models, which means that the generated text is dependent only on data that we can observe. A deterministic template is one which when executed twice on the same input performs the same actions and produces the same output, which means that we can always predict which parts of the input models the template will access. Under these conditions, property accesses alone can be used to determine whether or not a re-invocation of a template will produce a different output after the input models have been changed. A similar correctness argument is made for the incremental model consistency checking approach in [6].

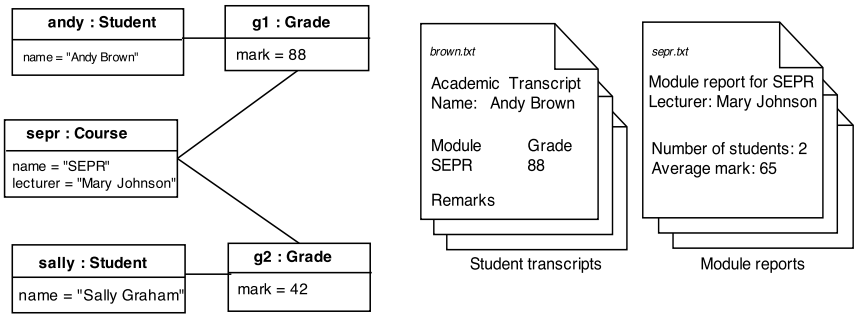
3.2 Design

In order to demonstrate the feasibility of *property access traces*, we extend EGL (the Epsilon Generation Language) [4]. EGL is a template-based M2T language. EGX is an orchestration sub-language of EGL which provides mechanisms for co-ordinating template execution.

Before discussing the details of implementing *property access* traces for EGL, we first describe the way in which transformation execution is implemented in the language (Figure 2). An M2T transformation in EGL is specified in the form of an EGX program, which comprises a number of rules and EGL templates. Typically, each rule will also contain a target, which is a specification of the destination of the output of the transformation. In its simplest form, a rule binds an EGL template to a metamodel type and executes the template for each model element of the correct type. The transformation engine starts by loading the input model(s), before executing the EGX program. Transformation execution begins by evaluating each rule in the EGX program, to determine its metamodel type, then invokes the associated *Template* on every model element of its type[1], and writes the output of executing the templates to files.

Consider, for example, the M2T transformation in Listing 1.2, which produces student transcripts and course reports of the forms shown on the right-hand side

[1] EGX rules also support *guards* which can further limit their applicability



(a) Model. (b) Example output.

Fig. 1. Artefacts for a M2T transformation that generates reports.

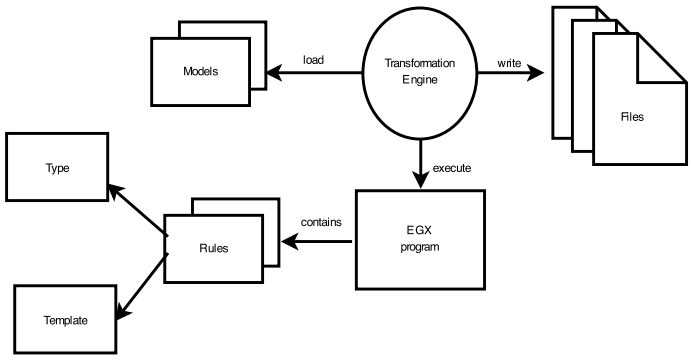


Fig. 2. Overview of transformation execution using EGX.

of Figure 1. This EGX program comprises two transformation rules: *StudentToTranscript* (lines 1 -5), *CourseToReport* (lines 7 -11). EGX passes each object of type *Student* to the *studentToTranscript.egl* template (Listing 1.3) and each object of type *Course* to the *courseToReport.egl* template (Listing 1.4). Additionally, in each transformation rule, a target (filename) is defined, whose value is determined at the transformation execution time.

In a typical M2T (batch) transformation engine, execution involves evaluating all templates against all instances of that context type every time a transformation is executed. In a source incremental M2T transformation engine, transformation execution involves identifying only the templates that need to be re-evaluated to propagate changes from the source model to the generated text. In other words, a source incremental M2T transformation engine identifies but, crucially, does not re-evaluate templates for which the generated text is known from a previous invocation of the transformation.

```
1  rule StudentToTranscript
2     transform aStudent : Student {
3        template : "studentToTranscript.egl"
4        target : aStudent.name + ".txt"
5  }
6
7  rule CourseToReport
8     transform aCourse : Course {
9        template : "courseToReport.egl"
10       target : aCourse.name + ".txt"
11 }
```

Listing 1.2. Example of an EGX M2T program applied to input model in Figure 1(a).

```
1  Student name : [%= aStudent.name %]
2  Course  Grade
3  [% for(grade in aStudent.grades) { %]
4  [%= grade.course.name + " " + grade.mark %]
5  [% } %]
```

Listing 1.3. M2T template for generating student transcripts specified in EGL syntax.

3.2.1 Extending M2T transformation languages with Property Access Traces. Implementation of property access traces involves extending the execution engine of a M2T language with four new concepts. Property access traces comprise transformation information that is derived from model elements and from the templates that are invoked on those model elements. During the execution of a template, a *PropertyAccessRecorder* captures the properties of the accessed model elements. The recorded *PropertyAccess(es)* which make up a *PropertyAccessTrace* are then persisted in non-volatile storage, a *PropertyAccessStore*. Figure 3 illustrates the conceptual organisation of the information contained in a *PropertyAccessTrace*.

- A ***PropertyAccess*** is a triple <e, p, v>, where e is the unique identifier of the model element, p is the name of the property, and v is the current value of the property. The way in which model element identifiers are computed varies, depending on the underlying modelling technology (e.g., XMI IDs or relative paths for EMF XMI models). There are two types of property accesses – *AttributeAccesses* and *ReferenceAccesses* – which vary in the type of value that they store. *AttributeAccesses* store a string value and are used when the property has a primitive type. *ReferenceAccesses* store the unique identifiers of the referenced model elements and are used when the property is a reference.
- A ***PropertyAccessTrace*** (Figure 3) captures which transformation rules are invoked on which source model elements and, moreover, which *PropertyAccesses* resulted from each invocation of a transformation rule (a *RuleInvocation)* in Figure 3).
- A ***PropertyAccessRecorder*** is responsible for recording *PropertyAccesses* during the execution of a template, and updating the *PropertyAccesses* when a change in the value of a *PropertyAccess* is detected. It is important to note that since property access traces contains data about input model elements only, any other type of change to the transformation specification is not

```
1  Course Report for [%= aCourse.name %]
2  Lecturer: [%= aCourse.lecturer %]
3
4  Number of students:[%= aCourse.grades.size() %]
5  Average mark:[%=aCourse.grades.collect(mark).sum()/aCourse.grades.size() %]
```

Listing 1.4. M2T template for generating course reports specified in EGL syntax

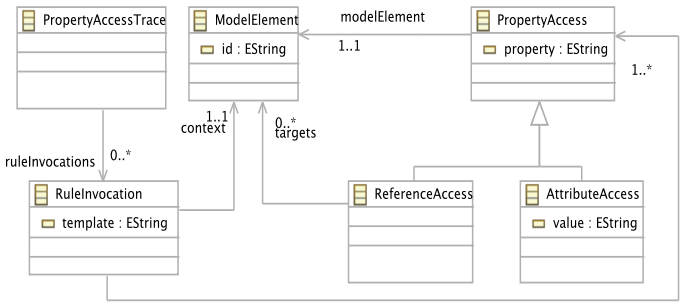


Fig. 3. Overview of Property Access Trace.

considered (See section 4.3 for a discussion on known limitations of this approach).

– A ***PropertyAccessStore*** is responsible for storing the *PropertyAccess*es passed on to it by the *PropertyAccessRecorder*. The *PropertyAccessStore* is also responsible for making *PropertyAccess*es (that were stored during a previous transformation execution) available to the transformation engine. We use an embedded RDBMS to store *property access*es, but other options (e.g., graph databases, XML documents, etc.) are also possible. A *PropertyAccessStore* must be capable of persisting, in non-volatile storage, the property access trace information between invocations of a M2T transformation. The main requirement for a *PropertyAccessStore* is performance: any gains achieved with a source incremental engine might be negated if the *PropertyAccessStore* cannot efficiently read and write property access traces.

We now briefly describe the way in which these concepts are used to achieve source incremental transformation, before providing an example. During the initial execution of a transformation, the *PropertyAccessRecorder* creates *PropertyAccesses* from the properties of model elements that are accessed during the execution of each rule. The collected *PropertyAccesses* are organised by *RuleInvocation* by the transformation engine to form a *PropertyAccessTrace* and stored by the *PropertyAccessStore*. At subsequent execution of the M2T transformation, the transformation engine retrieves the previous *PropertyAccessTrace* from the *PropertyAccessStore*. Whenever the transformation engine would ordinarily invoke a transformation rule, it instead retrieves each relevant *PropertyAccess* from the *PropertyAccessTrace* and queries the model to determine if the value of any of the *PropertyAccess*es has changed. Only when a value has changed is the

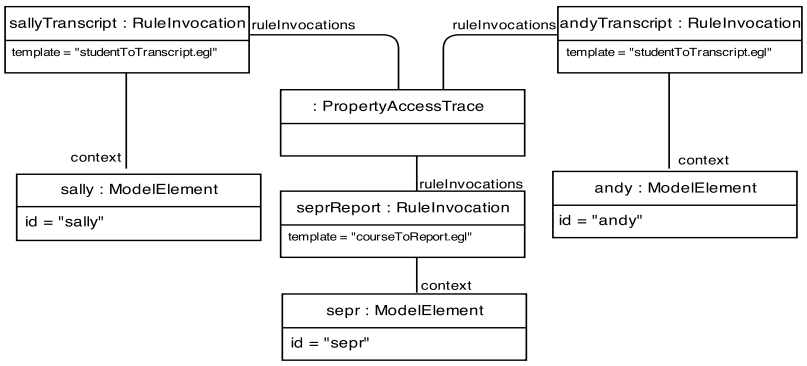


Fig. 4. A partial property access trace for executing *studentToTranscript.egl* on *andy* and *sally*, and *courseToReport.egl* on *sepr*.

transformation rule invoked. The *PropertyAccessTrace* is updated and stored if any values have changed.

3.2.2 Example. To further demonstrate the way in which property access traces achieve source-incremental M2T transformation, we now consider an example. Our example uses the transformation in Listings 1.3 and 1.4, which generate student transcripts and course reports from a university model. Executing the transformation on the simple university model in Figure 1(a) causes the transcript-generating rule to be invoked once on each student, *andy* and *sally*, and the course report-generating rule once on course *sepr*. As such, the resulting property access trace comprises three rule invocation objects (Figure 4). Each rule invocation object comprises several property accesses, which are recorded during the execution of the templates in Listing 1.3 and 1.4.

Let us consider the properties accessed during the invocation of the template on *sally*. The *sallyTranscript* rule invocation (Figure 5) comprises several attribute and reference access objects and is constructed as follows. Firstly, the template accesses `sally.name` (line 1 of Listing 1.3) and creates the *aa1* attribute access (Figure 5). The template then accesses `sally.grades` (line 3) and this creates the *ra1* reference access. The *grade.course.name* traversal expression in the template (line 4) creates two property accesses: the *ra2* reference access for *grade.course* and the *aa2* attribute access for *course.name*. Finally, the *grade.mark* expression (line 5) creates the *aa3* attribute access. The boxes with a dashed border in Figure 5 reinforce the relationship between property access objects in the trace and the expressions in the template (Listing 1.3). Note that each property access stores a reference to the model element from which its value was obtained.

When the M2T transformation is executed again, the transformation engine retrieves the property access trace (including Figures 4 and 5) and queries the parts of the model that were previously accessed by the transformation, such as

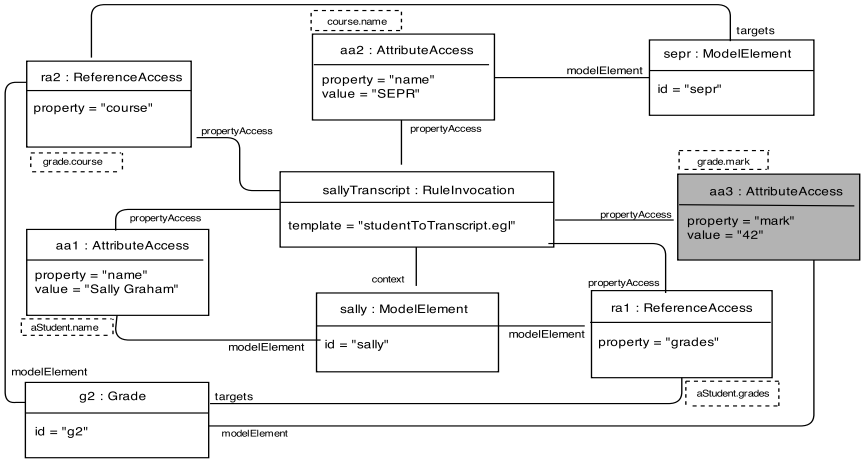


Fig. 5. Expansion of the property access trace for the *sallyTranscript* rule invocation.

the name of each student. Only when the value of any property differs from the value stored in a property access is the containing rule invocation re-executed.

For example, the *sallyTranscript* rule invocation (Figure 5) indicates that if all of the following constraints hold, then the rule invocation need not be re-executed:

1. `sally.name == "Sally Graham"` – due to *aa1*
2. `sally.grades == {g2}` – due to *ra1*
3. `g2.course == sepr` – due to *ra2*
4. `sepr.name == "SEPR"` – due to *aa2*
5. `g2.mark == "42"` – due to *aa3*

Suppose that Sally's grade for the SEPR course is changed: the mark attribute of *g2* is changed from 42 to 54. Note that the *aa3* attribute access (highlighted in Figure 4) stores the old value for the mark, 42. When the transformation is re-executed, condition #5 above will no longer hold: *g2.mark* will now evaluate to 54. Consequently, the transformation engine will re-execute the *sallyTranscript* rule invocation.

We have not shown the complete property access trace for the *andyTranscript* rule invocation (due to space constraints), but it is very similar in structure to the *sallyTranscript* rule invocation in Figure 5. The property accesses for *andyTranscript* result in the following constraints:

1. `andy.name == "Andy Brown"`
2. `andy.grades == {g1}`
3. `g1.course == sepr`
4. `sepr.name == "SEPR"`
5. `g1.mark == "88"`

From these constraints, it is clear that the change to *g2.mark* does not require a re-execution of the *andyTranscript* rule invocation as none of the constraints above depend on *g2.mark*. If, on the other hand, our change had been to *sepr.name* rather than to *g2.mark*, then both of the sets of constraints shown above would be unsatisfied and both the *sallyTranscript* and the *andyTranscript* rule invocations would be re-executed.

In general, determining whether or not a rule invocation needs to be re-executed requires the evaluation of *O(n)* constraints where n is the number of property accesses for that rule invocation.

4 Evaluation and Experience Report

In this section we report on the results of the empirical evaluation of *property access traces*, in which we compare the transformation execution times in incremental and non-incremental modes for two existing transformations. The results of our experiment show that source incremental transformations can be more efficient than non-incremental transformations, particularly for frequent or relatively small changes to models.

4.1 Empirical Evaluation

To assess the performance of *property access traces*, we used two existing EGL transformations: Pongo and GraphitiX. We investigated whether property access traces are effective when used for repeated invocations of code generation over the lifetime of an MDE project (Pongo), and whether property access traces are effective as the proportion of change in the input model increases (GraphitiX). We also investigated the memory and disk usage of *property access traces* (Pongo and GraphitiX) to ensure that resource usage is reasonable.

4.1.1 Pongo Pongo[2] generates data mapper layers for MongoDB, a non-relational database. Pongo takes as input an Ecore model that describes the types and properties of the objects to be stored in the database, and generates Java code that can be used to interact with the database via the user-defined types and properties (without needing to use the MongoDB API).

We compared the total time taken for incremental and non-incremental code generation over the lifetime of a real MDE project. For this purpose we used Pongo v0.5, and 11 versions of the GmfGraph Ecore model (obtained from the Git repository[3] of the GMF team). To simulate code generation activities in the GMF project, we ran Pongo using non-incremental and incremental EGL on each version of the GmfGraph Ecore model.

The results (Table 1) show the difference in number of template invocations and total execution time between non-incremental and incremental execution

[2] `https://code.google.com/p/pongo/`
[3] `https://git.eclipse.org/c/gmf-tooling`

		Non-Incremental		Incremental	
Version	Changes (#)	Invocations (#)	Time (s)	Invocations (#)	Time (s; %)
1.23	-	72	1.79	72	2.29 (128%)
1.24	1	73	1.72	6	0.93 (54%)
1.25	1	73	2.01	4	0.69 (34%)
1.26	1	74	2.03	6	0.66 (33%)
1.27	10	74	1.97	44	0.78 (40%)
1.28	10	74	1.95	44	0.67 (34%)
1.29	14	74	1.94	14	0.52 (27%)
1.30	24	77	2.02	41	0.70 (35%)
1.31	1	77	1.86	0	0.40 (22%)
1.32	1	77	1.95	0	0.38 (19%)
1.33	3	79	2.00	8	0.55 (28%)
Total			21.24		8.57 (40%)

Table 1. Results of using non-incremental and incremental M2T transformation for the Pongo M2T transformation, applied to 11 historical versions of the GMFGraph Ecore model.

modes of execution, for each of the 11 versions of the GmfGraph model. Expectedly, during the first invocation of the transformation (version 1.23) in incremental mode, the execution took slightly longer to execute than the non-incremental mode because the former incurs an overhead as the transformation in addition to evaluating templates, must record and process model element properties that are accessed in each template. However, during subsequent executions of the transformation, the incremental mode of execution required between 19% and 54% of the execution time required by the non-incremental mode. In other words, during the execution of the transformation on all versions of the GmfGraph project, we observed upto an 81% reduction in total execution time. Although the overall reduction in execution time (12.67s) is modest, that is partly explained by the relatively small size of the Pongo transformation (6 EGL templates totalling 329 lines of code), and of the GmfGraph model (averaging 65 classes).

4.1.2 GraphitiX GraphitiX[4] is a Java code generator for Graphiti-based graphical model editors. GraphitiX takes as input annotated Ecore models, which contain a description of the syntax of a domain-specific modelling language. GraphitiX (23 EGL templates totalling 1689 lines of code) is much larger than Pongo.

As such, we used GraphitiX to investigate whether property access traces are effective as the proportion of change in the input model increases. In particular, we sought to identify how large a change to the input model was necessary in order for incremental transformation to become slower than non-incremental transformation due to the overhead incurred in querying a property access trace.

For this purpose we used GraphitiX (Subversion revision 1) and a synthetic Ecore model. We executed GraphitiX on the model, made a change to the model,

[4] https://code.google.com/p/graphiti-x/

	Non-Incremental		Incremental	
Changes (Elements #; %)	Templ. Invocations (#)	Time (s)	Templ. Invocations (#; %)	Time (s)
-	4014	14.13	4014	20.63
1 (0.1%)	4014	12.09	9 (0.22%)	7.85
5 (0.5%)	4014	14.44	25 (0.62%)	6.92
10 (1%)	4014	14.09	45 (1.12%)	7.59
20 (2%)	4014	13.86	85 (2.11%)	7.13
100 (10%)	4014	15.01	405 (10.09%)	8.60
300 (30%)	4014	14.83	1205 (30.01%)	11.35
600 (60%)	4014	14.30	2405 (59.92%)	16.30
700 (70%)	4014	14.44	2805 (69.88%)	18.50

Table 2. Results of using non-incremental and incremental M2T transformation for the GraphitiX M2T transformation, applied to increasingly larger proportions of changes to the source model.

and re-executed GraphitiX in incremental and non-incremental modes. We varied the proportion of change made to the model. We changed the model by modifying a subset of all classes (by renaming the class and one of its attributes). We chose this type of modification because, as developers of GraphitiX, we knew that that the transformation would be sensitive to these changes. Table 2 supports this claim: the proportion of template invocations in incremental mode is roughly the same as the proportion of change made to the model. In other words, we selected this type of modification to avoid changing the model in a way that had very little impact on the generated artefacts, or vice versa.

As shown in Table 2, our results suggest that source incremental transformation using *property access traces* requires less computation until a significant proportion (threshold) of the input model is changed. In this case, that threshold was reached when approximately 60% of the input model was changed (see the highlighted row in Table 2). This corresponds to 1200 changes, as 2 changes were applied to each changed model element. The threshold will be different for other transformations, and will depend on factors such as: the amount of *property accesses* in templates, and the complexity of model queries in the templates.

4.1.3 Memory and Disk Utilization To demonstrate that our approach is feasible with respect to resource usage, we investigated the memory and disk usage of property access traces during our experiments with Pongo and GraphitiX.

With respect to memory usage, we observed that peak memory usage for incremental EGL was slightly higher than for non-incremental EGL. For Pongo, peak memory usage for incremental EGL was 102% of non-incremental EGL (200.1Mb compared to 196.7Mb). For GraphitiX, peak memory usage for incremental EGL was 110% of non-incremental EGL (480Mb compared to 436Mb).

With respect to disk usage, we observed that property access traces have modest requirements particularly for a modern development machine: the average size of the property access trace on disk was 412Kb for Pongo and 6.9Mb for GraphitiX. We have not yet optimised our implementation of property access traces to reduce disk space usage.

It is important to note that the memory and disk usage will vary for different transformations, depending on the size of the input model and in particular the amount of *property accesses* made by the transformation.

4.2 Discussion

Our initial experiments indicate that the use of property access traces for providing source incrementality is promising: we have demonstrated that a reduction in execution time is observed for realistic changes to a model (e.g., the changes made to GmfGraph Ecore model). The results also indicate that source incrementality using our approach is more efficient than non-incremental transformations when frequent, small changes are made to a model throughout the lifetime of a project.

The results of the experiments in our previous work [7], which used *signatures* for source-incremental M2T transformation suggested that source incrementality can be used to realize upto 45% performance gain in transformation execution time. *Property access* traces offer a further 15% reduction in transformation execution time. Overall, a 60% reduction in transformation execution time was observed using *property access* traces.

It is important to note that the example M2T demonstrated in section 3.2.2 was simplified for brevity. The templates (in Listings 1.3 and 1.4) did not contain model-querying statements such as, collection-filtering operations (e.g., `Student.all.select(s|s.name == "Andy Brown")`). However, as discussed in [6] and in Section 3.1, the complexity of navigation expressions (as long as they are deterministic) is irrelevant. *The two M2T transformations we used for evaluating our approach make extensive use of complex OCL-like collection navigation and filtering operations.*

Lastly, an incremental M2T transformation is correct if it results in the regeneration of all the required files whose contents were affected by the change(s) to the input model. To verify the correctness of the incremental execution of the two transformations that we used in the evaluation of *property access traces*, we performed tests which compared the output of the transformations in incremental mode with the output of the transformations in non-incremental mode. The outcome of our tests indicate that the contents of the files generated in incremental mode were always the same as the contents of the same files generated in non-incremental mode.

4.3 Limitations of Property Access Traces

Property access traces exhibit some limitations. Some of these limitations relate to our current implementation – and will be addressed in future work – whilst some limitations are inherent to the approach.

Our current implementation of *property access traces* in EGL monitors property accesses only during the execution of templates. However, *property access traces* can become over-sensitive to changes to parameters contained in unordered collections because it cannot distinguish between unordered and ordered

collections. Consider a template (e.g., `[%= Student.grades.mark %]`) that only prints out the grades of a student, the *PropertyAccessRecorder* records a property access of *grades* on *Student*, whose value is a collection of *Grades*, and also records a property access of *mark* on each *Grade* in the collection *Student.grades*. If in a change event, a *Grade* is removed and re-added to the collection *Student.grades*, these modification operations will result in the same set of *Student.grades*, albeit with a different order, since the re-added *Grade* is inserted at the back of the collection. This will cause the template to be re-executed unnecessarily. The order of collections are important for accurate comparison of modified structural features of a model element. Our current implementation does a string comparison of the values of *property accesses* recorded from calls that return a collection of structural features, and cannot detect if mere reordering of collections is a significant change event. Furthermore, our current implementation does not record the accessing of all model elements of a specific type (e.g. Student.allInstances().size()). We currently have a prototypical implementation that extends our *PropertyAccessRecorder* in order to record accesses of *allInstances* nature.

Inherent limitations of the *property access trace* approach are that the use of non-deterministic programming constructs (e.g., random number generators, hash-sets, hash-maps) in a template prevent source incrementality (because the template must always be invoked to compute an appropriate result), and that property access traces can be pessimistic: it is conceivable that a template might access a property but not use its value in the generated text (e.g., `[% if(aGrade.mark > 70) { //do-nothing } %]`). In these cases, a property access trace would result in an unnecessary re-execution of the template.

5 Related Work

Property access traces follow the model profiling method for model consistency checking by Egyed [6]. However, our approach differs in the sense that *property access traces* detect input model changes at runtime, while Egyed's approach assumes notifications of input model changes (e.g., from the modelling technology). Our approach does not rely on the model editor or the underlying modelling framework, hence it can be readily applied to a new version of an input model to compute model changes, with respect to the transformation, as shown in the running example (Section 3.2.2).

To the best of our knowledge, Xpand[5] is the only contemporary M2T language that supports source incremental transformation. Incremental generation in Xpand uses a combination of trace links and model differencing techniques. Difference models are used to determine changed subset of input models, and trace links are used to specify how source model elements are mapped to generated files. Once the difference model is constructed, impact analysis is performed to determine which changed model elements are used in which templates. A

[5] `http://eclipse.org/modeling/m2t/?project=xpand`

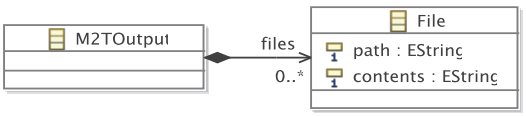


Fig. 6. Example representation of a M2T as M2M metamodel.

template is re-executed if it consumes a model element that has changed. The efficiency of the approach to incrementality employed by Xpand is heavily dependent on the effectiveness of the underlying modelling framework in performing model differencing. For instance, calculating model diffs between all the versions of GmfGraph models used for the Pongo transformation took about 1.3 seconds (average) using EmfCompare which is the same tool that Xpand uses to compute model diffs. This figure represents the time taken to perform only a part of the computation done by Xpand's incremental engine exceeds the time taken to execute each Pongo transformation (see Table 1) all versions of the GmfGraph model. As model differencing is integral to Xpand's incremental method, there is no need to conduct a full scale comparison of *property access traces* and model differencing incremental approaches. Furthermore, performance can be impaired because model differencing requires that (at least) two versions of the input model, along with a diff model are loaded, which requires at least three model traversals. This might also be impractical since access to the previous version of the model is needed and may not be available. Property access traces as explained in section 3 do not require model differencing and hence offer a fundamentally different approach to source incrementality.

An M2T transformation could be expressed as an M2M transformation, where the target metamodel resembles Figure 6. As such, a Turing-complete incremental M2M language could be used to express incremental M2T transformations. Song et. al. use model differencing and static analysis to incrementally execute QVTr transformations [8], and as such have the same limitations as Xpand, discussed above. Additionally, static analysis can be too pessimistic to be useful for incremental transformation as discussed in our previous work [7]. More generally, M2M languages may be limited in their ability to handle unique features of M2T languages (e.g., handling protected regions, white spaces, escape direction, etc.). There has been considerable work on incrementality for triple-graph grammars – see [9] for a recent comparison – but TGGs are generally not Turing-complete (although some do provide fallback mechanisms). Additional research is needed to investigate whether the restricted capabilities of incremental TGGs are sufficient to implement complex model transformations.

6 Conclusion

Despite the potential productivity and portability gains of MDE, the inability of MDE tools and techniques to support the building of large and complex systems through processes that scale remains an open research challenge. In this paper, we proposed *property access traces*, an approach to reducing the execution time

of M2T transformations in response to changes to source models. We have contributed a design for extending M2T transformation languages with support for *property access traces*, and demonstrated the feasibility of *property access traces* through an empirical evaluation. We have shown that the potential performance gains of source incremental transformation via property access traces are substantial: we observed an average reduction in transformation execution time of 60%. Instead of computing model differences between versions of input models as used by Xpand's incremental transformation technique, *property access traces* employs a technique that only requires the current state of a model, whose efficiency also does not depend on the effectiveness of an underlying modelling framework to calculate model diffs.

In future work, we will improve our implementation of property access traces to address the limitations described in Section 4.3, after which we will extend our empirical evaluation to investigate incrementality for larger and more complicated M2T transformations (such as the INESS M2T transformation described in Section 1).

Acknowledgements. This work was partially supported by the European Commission, through the Scalable Modelling and Model Management on the Cloud (MONDO) FP7 STREP project (grant #611125). The motivating example discussed in this paper was taken from Rose's work on the INESS project, which was supported by the European Commission and co-funded under the 7th Framework Programme (grant #218575).

References

1. P. Mohagheghi et al. MDE adoption in industry: challenges and success criteria. In *Models in Software Engineering*, volume 5421, pages 54–59. Springer, 2009.
2. D. Kolovos et al. Scalability: The holy grail of Model Driven Engineering. In *ChaMDE 2008 Workshop Proceedings*, pages 10–14, 2008.
3. K. Czarnecki and S. Helsen. Feature-based survey of model transformation approaches. *IBM Systems Journal*, 45(3):621–645, 2006.
4. L.M. Rose et al. The Epsilon Generation Language. In *Proc. ECMDA-FA*, volume 5095 of *LNCS*, pages 1–16. Springer, 2008.
5. B. Ogunyomi. Incremental model-to-text transformation (qualifying dissertation). Technical report, 2013.
6. Alexander Egyed. Automatically Detecting and Tracking Inconsistencies in Software Design Models. *Software Engineering, IEEE Transactions on*, 37(2):188–204, 2011.
7. B. Ogunyomi et al. On the use of signatures for source incremental model-to-text transformation. In *MoDELS*, volume 8767 of *LNCS*, pages 84–98. Springer, 2014.
8. Song, Hui and Huang, Gang and Chauvel, Franck and Zhang, Wei and Sun, Yanchun and Shao, Weizhong and Mei, Hong. Instant and incremental QVT transformation for runtime models. In *Model Driven Engineering Languages and Systems*, pages 273–288. Springer, 2011.
9. Leblebici, Erhan and Anjorin, Anthony and Schürr, Andy and Hildebrandt, Stephan and Rieke, Jan and Greenyer, Joel. A Comparison of Incremental Triple Graph Grammar Tools. *Electronic Communications of the EASST*, 67, 2014.